网络地址转换实验报告

- 实验内容

  实验的主要难点在于 NAT 的翻译工作，其中首先需要判断 NAT 的数据流向，是从 私网发送 数据到公网，即私网的客户端向公网的服务器等请求消息，或者从公网的 客户端 向私网的服务器段请求消息。

```
// determine the direction of the packet, DIR_IN / DIR_OUT / DIR_INVALID
static int get_packet_direction(char *packet)
{
    fprintf(stdout, "TODO: determine the direction of this packet.\n");
    struct iphdr *packet_iphdr;
    packet_iphdr = (struct iphdr *)(packet + sizeof(struct ether_header));

    u32 ip1,ip2;
    ip1 = htonl(packet_iphdr-> saddr);
    ip2 = htonl(packet_iphdr -> daddr);

    rt_entry_t *packet_rtable1 = longest_prefix_match(ip1);
    rt_entry_t *packet_rtable2 = longest_prefix_match(ip2);

    iface_info_t *packet_iface1 = packet_rtable1 -> iface;
    iface_info_t *packet_iface2 = packet_rtable2 -> iface;

    if(nat.internal_iface == packet_iface1 && nat.external_iface ==
packet_iface2 ){
        return DIR_OUT;
    }
    else if(nat.external_iface -> ip == ip2 && nat.external_iface ==
packet_iface1){
        return DIR_IN;
    }
    return DIR_INVALID;
}
```

上面的操作都很自然，除了最后对于 nat 的 DIR_IN 的判断，由于我们假设现在的拓扑为

(P1 私网) P2（NAT) P3(公网)，

那么此时对于 DIR_IN 方向的数据流动，由于给定信息只是 P3 -> P2,所以此时需要分清

这时的 external_iface 所指向的 应该就是 ip2(也就是 daddr),表示NAT能够表达的源地址,而此时转发该消息的 port,也是 P3 通过最长前缀匹配得到的 port，是P2的某一个端口，这里是理解上的一个难点。

```
void do_translation(iface_info_t *iface, char *packet, int len, int dir)
{
    fprintf(stdout, "TODO: do translation for this packet.\n");
    struct iphdr *packet_iphdr;
    packet_iphdr = (struct iphdr *)(packet + sizeof(struct ether_header));
    struct tcphdr *packet_tcphdr;
    packet_tcphdr = (struct tcphdr *)(packet + sizeof(struct ether_header) +
IP_HDR_SIZE(packet_iphdr));
    u32 ips,ipd;
```

```
    ips = htonl(packet_iphdr-> saddr);
    ipd = htonl(packet_iphdr -> daddr);
    u16 ports,portd;
    ports = htons(packet_tcphdr -> sport);
    portd = htons(packet_tcphdr -> dport);

    rt_entry_t *packet_rtables = longest_prefix_match(ips);
    rt_entry_t *packet_rtabled = longest_prefix_match(ipd);

    iface_info_t *packet_ifaces = packet_rtables -> iface;
    iface_info_t *packet_ifaced = packet_rtabled -> iface;

    if(dir == DIR_IN){
        u32 packet_remote_ip;
        u16 packet_remote_port;

        packet_remote_ip = ips;
        packet_remote_port = ports;

        u32 packet_external_ip;
        u16 packet_external_port;

        packet_external_ip = ipd;
        packet_external_port = portd;

        u8 hash_code = hash8(&packet_remote_ip, 4) ^
hash8(&packet_remote_port,2);
        struct list_head *packet_list_head;
        packet_list_head = &nat.nat_mapping_list[hash_code];
        struct nat_mapping *req_entry,*req_q;
        list_for_each_entry(req_entry,packet_list_head,list){
            if(req_entry -> remote_ip == packet_remote_ip && req_entry ->
remote_port == packet_remote_port \
            && req_entry -> external_ip == packet_external_ip && req_entry -
> external_port == packet_external_port){
                packet_iphdr->daddr = htonl(req_entry -> internal_ip);
                packet_tcphdr->dport = htons(req_entry -> internal_port);
                packet_tcphdr -> checksum =
tcp_checksum(packet_iphdr,packet_tcphdr);
                packet_iphdr -> checksum = ip_checksum(packet_iphdr);
                ip_send_packet(packet,len);
                return;
            }
        }

        struct dnat_rule *req_temp, *req_p;
        list_for_each_entry(req_temp,&(nat.rules),list){
            if(req_temp->external_ip == packet_external_ip && req_temp-
>external_port == packet_external_port){
                struct nat_mapping *target;
                target = (struct nat_mapping *)malloc(sizeof(struct
nat_mapping));
                target -> remote_ip = packet_remote_ip;
                target -> remote_port = packet_remote_port;
                target -> internal_ip = req_temp -> internal_ip;
                target -> internal_port = req_temp -> internal_port;
                target -> external_ip = packet_external_ip;
                target -> external_port = packet_external_port;
```

```
                target -> update_time = time(NULL);
                list_add_tail(&(target->list), &
(nat.nat_mapping_list[hash_code]));

                packet_iphdr->daddr = htonl(req_temp -> internal_ip);
                packet_tcphdr->dport = htons(req_temp -> internal_port);
                packet_tcphdr -> checksum =
tcp_checksum(packet_iphdr,packet_tcphdr);
                packet_iphdr -> checksum = ip_checksum(packet_iphdr);
                ip_send_packet(packet,len);
                return;
            }
        }
    }
    else if(dir == DIR_OUT){
        u32 packet_remote_ip;
        u16 packet_remote_port;

        packet_remote_ip = ipd;
        packet_remote_port = portd;

        u32 packet_internal_ip;
        u16 packet_internal_port;

        packet_internal_ip = ips;
        packet_internal_port = ports;

        u8 hash_code = hash8(&packet_remote_ip, 4) ^
hash8(&packet_remote_port,2);
        struct list_head *packet_list_head;
        packet_list_head = &nat.nat_mapping_list[hash_code];
        struct nat_mapping *req_entry,*req_q;
        list_for_each_entry(req_entry,packet_list_head,list){
            if(req_entry -> remote_ip == packet_remote_ip && req_entry ->
remote_port == packet_remote_port \
            && req_entry -> internal_ip == packet_internal_ip && req_entry -
> internal_port == packet_internal_port){
                packet_iphdr->saddr = htonl(req_entry-> external_ip);
                packet_tcphdr -> sport = htons(req_entry -> external_port);
                packet_tcphdr -> checksum =
tcp_checksum(packet_iphdr,packet_tcphdr);
                packet_iphdr -> checksum = ip_checksum(packet_iphdr);
                ip_send_packet(packet,len);
                return;
            }
        }

        for(int i = NAT_PORT_MIN; i < NAT_PORT_MAX; i++){
            if(nat.assigned_ports[i] == 0){
                nat.assigned_ports[i] = 1;
                struct nat_mapping *target;
                target = (struct nat_mapping *)malloc(sizeof(struct
nat_mapping));
                target -> remote_ip = packet_remote_ip;
                target -> remote_port = packet_remote_port;
                target -> internal_ip = packet_internal_ip;
                target -> internal_port = packet_internal_port;
```

```
                target -> external_ip = nat.external_iface-> ip;
                target -> external_port = i;

                target -> update_time = time(NULL);
                list_add_tail(&(target->list), &
(nat.nat_mapping_list[hash_code]));
                packet_iphdr->saddr = htonl(target -> external_ip);
                packet_tcphdr -> sport = htons(target  -> external_port);
                packet_tcphdr -> checksum =
tcp_checksum(packet_iphdr,packet_tcphdr);
                packet_iphdr -> checksum = ip_checksum(packet_iphdr);
                ip_send_packet(packet,len);
                return;
            }
        }
    }
}
```

接下来的翻译比较复杂，但大多数操作可以直接复制粘贴得到。首先对于 in 方向，如果存在连接，则直接修改 ip 和 tcp 头部的checksum，否则需要根据公网到私网的建立规则，分配空间，将新分配的内容插入到表中。

如果是out方向，有连接的部分和in方向的类似，但没有连接的部分，则不具有规则可以借用。所以需要从 assigned_ports 中动态分配端口。

确认连接可以使用简单的hash8进行查找，但是在匹配的时候，需要4项都进行匹配。

```
// nat timeout thread: find the finished flows, remove them and free port
// resource
void *nat_timeout()
{
    while (1) {
        fprintf(stdout, "TODO: sweep finished flows periodically.\n");
        sleep(1);
        for(int i = 0; i < HASH_8BITS; i++){
            struct nat_mapping *req_entry,*req_q;

list_for_each_entry_safe(req_entry,req_q,&nat.nat_mapping_list[i],list){
                if(time(NULL) - req_entry-> update_time >
TCP_ESTABLISHED_TIMEOUT || is_flow_finished( &(req_entry -> conn) )){
                    u16 return_port = req_entry -> external_port;
                    int flag = 0;
                    struct dnat_rule *req_temp, *req_p;
                    list_for_each_entry(req_temp,&(nat.rules),list){
                        if(req_temp->external_port == req_entry-
>external_port &&  req_temp->external_ip == req_entry->external_ip && \
                            req_temp -> internal_port == req_entry ->
internal_port && req_temp -> internal_ip == req_entry-> internal_ip){
                            flag = 1;
                            break;
                        }
                    }
                    if(flag == 0) nat.assigned_ports[return_port] = 0;
                    list_delete_entry(req_entry);
                    free(req_entry);
                }
            }
        }
```

```
    }
```

需要注意，不出现段错误地删除，需要利用 list_for_each_entry_safe。同时在释放端口的时候，需要根据端口是否为 公网向私网接入的端口(也就是需要根据 rules来遍历查找)，如果是这种情况，则端口不能被释放。

```
void nat_exit()
{
    fprintf(stdout, "TODO: release all resources allocated.\n");
    for(int i = 0; i < HASH_8BITS; i++){
        struct nat_mapping *req_entry,*req_q;

list_for_each_entry_safe(req_entry,req_q,&nat.nat_mapping_list[i],list){
            list_delete_entry(req_entry);
            free(req_entry);
        }
        struct dnat_rule *req_temp, *req_p;
        list_for_each_entry_safe(req_temp,req_p,&(nat.rules),list){
            list_delete_entry(req_temp);
            free(req_temp);
        }
    }
    nat.internal_iface = NULL;
    nat.external_iface = NULL;
}
```

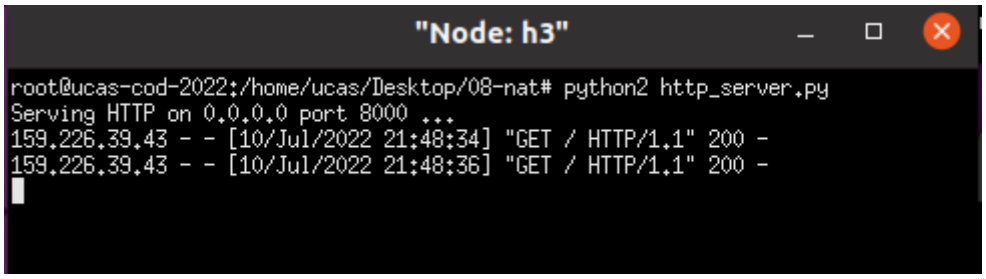
这里属于简单的释放操作，先 list_delete_entry,再 free掉。

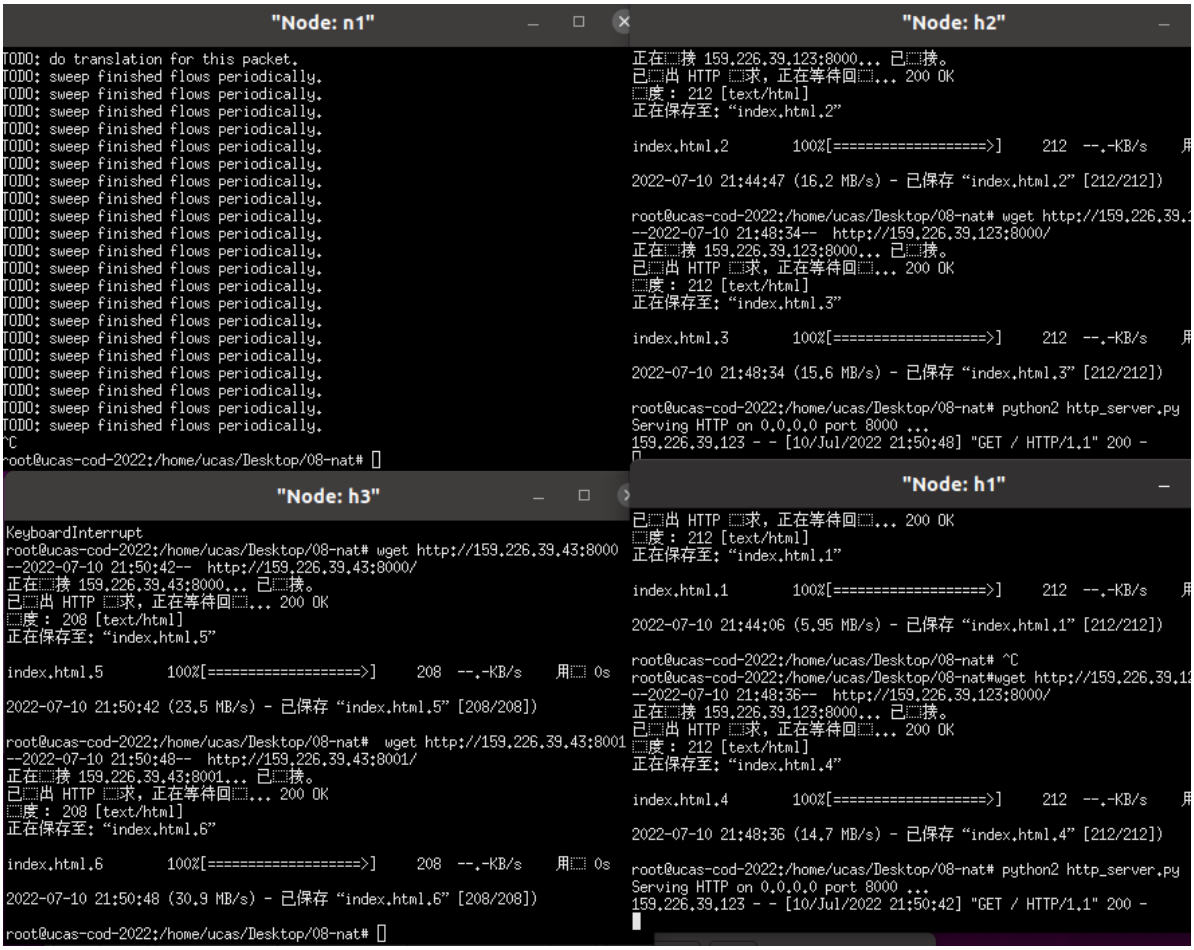
```
int parse_config(const char *filename)
{
    fprintf(stdout, "TODO: parse config file, including i-iface, e-iface
(and dnat-rules if existing).\n");
    FILE *fd;
    fd = fopen(filename,"r");
    char *buffer1;
    buffer1 = (char *)malloc(1024);
    fscanf(fd,"internal-iface: %s\n",buffer1);
    nat.internal_iface = if_name_to_iface(buffer1);
    char *buffer2;
    buffer2 = (char *)malloc(1024);
    fscanf(fd,"external-iface: %s\n",buffer2);
    nat.external_iface = if_name_to_iface(buffer2);
    int n1,n2,n3,n4,n5;
    int d1,d2,d3,d4,d5;
    int c;
    while((c = fscanf(fd,"dnat-rules: %d.%d.%d.%d:%d ->
%d.%d.%d.%d:%d\n",&n1,&n2,&n3,&n4,&n5, \
    &d1,&d2,&d3,&d4,&d5))!=-1){
        struct dnat_rule *temp;
        temp = (struct dnat_rule *)malloc(sizeof(struct dnat_rule));
        temp -> external_ip = (n1 << 24) + (n2 << 16) + (n3 << 8) + n4;
        temp -> external_port = n5;
        temp -> internal_ip = (d1 << 24) + (d2 << 16) + (d3 << 8) + d4;
        temp -> internal_port = d5;
        list_add_tail(temp,&(nat.rules));
```

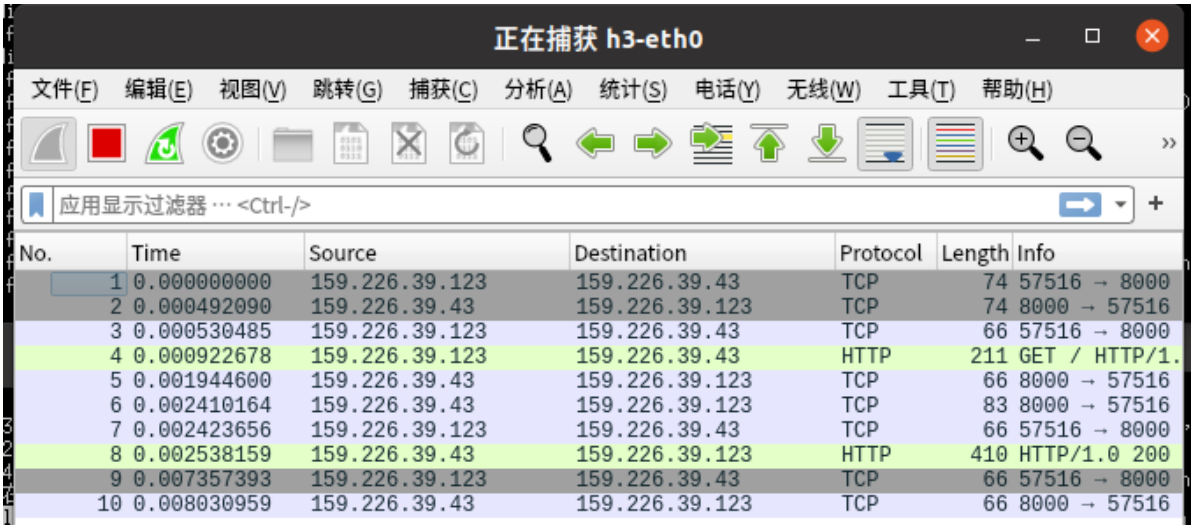
```
        fprintf(stdout,"%d %x %d %x %d\n", c, temp -> external_ip, temp ->
external_port, temp -> internal_ip, temp -> internal_port);
    }

    return 0;
}
```

对于文件的处理，需要特别注意 %s遇到空格不读取的情况。fscanf的返回值为 - 1 表示什么都没有读到，可以作为 while 循环的判断条件。

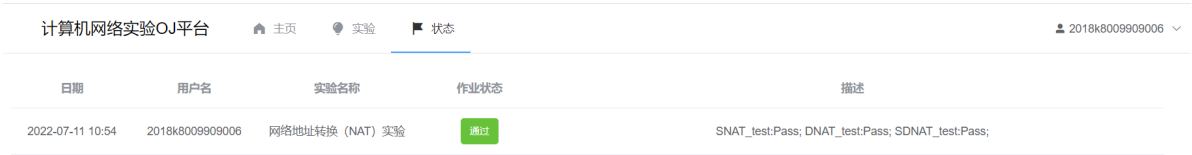
- 实验结果







- oj 运行结果



思考题

1.请调研说明NAT系统如何支持ICMP协议。

解:例如：局域网内的A：192.168.0.2/24 B:192.168.0.3/24

路由器两块网卡：192.168.0.1/24 188.10.1.2（公网地址）Intenet上web 服务器C： 200.10.2.1

比如A想访问C的web服务,便会发送HTTP请求，当HTTP请求达到路由器的时候，回进行SNAT转换,

这时候路由器上的NAT表回添加一行。NAT表通常是五元组，源IP和源端口就是发送HTTP请求的IP和端口，即A的IP和端口，而目的IP和目的端口则是指转换后的IP地址和端口，即进行SNAT后的源IP和源端口。当Web服务器C收到HTTP请求后，会发送响应报文。

当报文到达路由器后，会参照NAT表进行DNAT，参考依据，会在NAT表中查找目的IP和目的端口为188.10.1.2和54333的一行，然后用192.168.0.2和54333替换目的IP和目的端口。然后发往局域网内的A。通过浏览器呈现web网页。

以上就是局域网内机器访问外网的一个流程。从这个流程中可以得出，尽管只配置SNAT，但是在进行源地址和源端口转换的时候，会在路由器的NAT表中插入一行，当响应报文到达路由器的时候，参照NAT表中的信息，进行目的地址和目的端口的转换，所以在不配置DNAT的时候，目的地址和目的端口仍然可以被转换。但是这个过程，只能是局域网内的用户发起请求，响应报文的目的地址和目的端口才会被转换为局域网地址。外网的主机是不能直接访问局域网中的主机的。

ping是基于ICMP的，是没有端口的，那么这样是怎么来实现的呢？？？

没有端口，那就创造端口，在A发送ICMP报文的时候，会根据（Type+Code）的值生成源端口号，根据Identifier的值生成目的端口号，在路由器上进行SNAT，源IP更改后ICMP报文中的Identifier会改变，记作IDENTIFIER。在web服务器C收到ICMP请求后，生成ICMP响应报文，响应报文中的（Type+Code）会作为源端口，IDENTIFIER作为目的端口报文到达路由器后，根据NAT表中，查询目的IP和目的端口为188.10.1.2和IDENTIFIER的信息。将目的IP和目的端口换为

192.168.0.2和（Type+Code）,这样报文就可以成功的到达A了。

所以ICMP报文的NAT大致是根据ICMP报文的字段，形成伪端口，然后根据TCP报文的流程来处理。

2.给定一个有公网地址的服务器和两个处于不同内网的主机，如何让两个内网主机建立TCP连接并进行数据传输。（提示：不需要DNAT机制）

解：P2P可以使得不同局域网的内网设备可以直连。P2P（peer to peer）点对点技术又称对等互联网络技术，是一种网络新技术，依赖网络中参与者的计算能力和带宽，而不是把依赖都聚集在较少的几台服务器上。P2P网络通常用于通过Ad Hoc连接来连接节点。这类网络可以用于多种用途，各种档案分享软件已经得到了广泛的使用（譬如迅雷、电驴）。P2P技术也被使用在类似VoIP等实时媒体业务的数据通信中。

P2P有个很重要的能力，内网穿透能力，具有这个能力后，不同私网的设备可以直接进行通信。
根据客户端的不同，客户端之间进行P2P传输的方法也略有不同，P2P主要有中继、逆向连接、打洞（hole punching）等技术。

P2P打洞技术依赖于通常防火墙和锥形NAT（cone NAT）允许正当的P2P应用程序在中间件中打洞且与对方建立直接连接的特性。

根据传输方式的不同，P2P打洞技术分为TCP打洞及UDP打洞技术。
假设Client A打算与Client B直接建立一个UDP通信会话。如果Client A直接给Client B的公网地址138.76.29.7:31000发送UDP数据，NAT B很可能会无视进入的数据（除非是Full Cone NAT），Client B往Client A直接发信息也类似。

为了解决上述问题，在Client A开始给Client B的公网地址发送UDP数据的同时，Client A给Server S发送一个中继请求，要求Client B开始给Client A的公网地址发送UDP信息。Client A往Client B的输出信息会导致NAT A打开一个Client A的内网地址与Client B的外网地址之间的通讯会话，Client B往Client A亦然。当两个方向都打开会话之后，Client A和Client B就能直接通讯，而无须再通过Server S了。

UDP打洞技术有许多有用的性质。一旦一个的P2P连接建立，连接的双方都能反过来作为“引导服务器”来帮助其他中间件后的客户端进行打洞，极大减少了服务器的负载。应用程序不需要知道中间件是什么（如果有的话），因为以上的过程在没有中间件或者有多个中间件的情况下也一样能建立通信链路。